

Projet Raytracing - Phase 1, Phase 2 et Phase 3

Programmation 3D

Mathis Duban M1 IMAGINE



Table des matières

1 Phase 1	4
1.1 rayTrace	4
1.2 Calcul de l'intersection pour une sphère	4
1.3 Calcul de l'intersection pour un plan	6
1.4 Calcul de l'intersection avec un carré	7
2 Phase 2	9
2.1 Implémentation du modèle de Phong	9
2.2 Implémentation des ombres dures	10
2.3 Implémentation des ombres douces	11
3 Phase 3	12
3.1 Application d'un matériau Miroir : Réflexion	12
3.2 Application d'un matériau Vitre : Réfraction	13
3.3 Calcul de l'intersection avec un mesh	16
3.4 Ajout d'une structure d'accélération : KdTree	17
3.4.1 Etape 1 du Kdtree : la boîte englobante	17
3.4.2 Etape 2 du Kdtree : Création de l'arbre	18
3.5 Bonus 1 : Ajout de textures	19
3.5.1 Ajout de texture pour un carré	20
3.5.2 Ajout de texture pour une sphère	21
3.5.3 Ajout de texture pour un mesh	22
3.6 Bonus 2 : Optimisation des calculs matriciels	24
3.7 Bonus 3 : Indicateur de progrès du rendu de la scène	24
3.8 Bonus 4 : Application d'une normal map sur une surface	25

Table des figures

1	Représentation graphique des différents cas pour Δ	5
2	Modification de la position et de la mise à l'échelle	5
3	Récupération de l'intersection la plus proche	5
4	Représentation de l'intersection avec un plan	6
5	Schéma des projections sur le plan afin de vérifier que le rayon soit dans le carré	7
6	Création de la scène du raytracing avec les intersections des carrés et sphères	8
7	Scènes avec couleurs des objets modifiés	8
8	Affichage de la scène en ajoutant les composantes une à une	10
9	Scène avec implémentation des ombres dures	10
10	Scène avec implémentation des ombres douces	11
11	Scène avec deux boules miroirs de réflexion	12
12	Scène avec deux boules vitres de réfraction	14
13	Scène 2 illustrant la réfraction d'une sphère	14
14	Scène avec de la réfraction et réflexion	15
15	Scène avec les boîtes englobantes des Kdtree	17
16	Schéma représentant un kdTree	18
17	rendu d'une scène contenant 2 mesh complexes avec 871414 et 78966 triangles en 30 secondes	18
18	Schéma représentant la procédure d'insertion de texture	19
19	Schéma sur la récupération de coordonnées sur un carré	20
20	Exemple 1 d'application de texture sur des carrés	20
21	Exemple 2 d'application de texture sur des carrés	21
22	récupération des coordonnées u v pour une sphère	21
23	Exemple d'application de texture sur des sphères	22
24	Exemple 1 d'application de texture sur des mesh	23
25	Exemple 2 d'application de texture sur des mesh	23
26	Capture d'écran du chargement des textures	24
27	Capture d'écran de la progression du rendu via la barre de progression	24
28	Exemple de texture et sa normal map associée	25
29	Exemple de texture et sa normal map associée	26

Ce projet consiste à réaliser une scène 3D rendu par un système de lancés de rayon (**Ray Tracing**).

1 Phase 1

1.1 rayTrace

Dans cette première partie, on souhaite réaliser la fonction **rayTrace**. Cette fonction est essentielle dans le projet du ray tracing car elle permet de détecter et récupérer les différentes intersections des objets rencontrés lors des lancés de rayon que nous effectuons. Pour mieux visualiser le fonctionnement de la fonction, lorsque nous envoyons un rayon et que ce dernier touche bien un objet(sphère, carré), nous retournons la couleur de ce dernier et sinon on laisse en blanc. La manière dont les intersections sont calculées dépendent de la forme rencontrée, nous avons donc calculé les intersections pour les deux formes distinctes : Shpère et Cube.

1.2 Calcul de l'intersection pour une sphère

Pour calculer l'intersection de la sphère, nous nous aidons des différentes équations suivantes. L'équation d'une sphère de centre $C(x_c, y_c, z_c)$ et de rayon R est donnée par :

$$(x - x_c)^2 + (y - y_c)^2 + (z - z_c)^2 = R^2$$

L'équation paramétrique d'une droite passant par $O(x_o, y_o, z_o)$ avec une direction $\vec{d}(d_x, d_y, d_z)$ est :

$$\vec{P}(t) = O + t\vec{d} = (x_o + td_x, y_o + td_y, z_o + td_z)$$

En substituant l'équation de la droite dans celle de la sphère, on obtient une équation quadratique en t :

$$At^2 + Bt + C = 0$$

où :

$$A = d_x^2 + d_y^2 + d_z^2, \quad B = 2(d_x(x_o - x_c) + d_y(y_o - y_c) + d_z(z_o - z_c)), \quad C = (x_o - x_c)^2 + (y_o - y_c)^2 + (z_o - z_c)^2 - R^2$$

La solution de cette équation est obtenue en résolvant le discriminant Δ :

$$\Delta = B^2 - 4AC$$

Les valeurs de t sont données par :

$$t_1 = \frac{-B - \sqrt{\Delta}}{2A}, \quad t_2 = \frac{-B + \sqrt{\Delta}}{2A}$$

Nous avons donc 3 cas différents en fonction de Δ :

- Si $\Delta = 0$, il y a une tangence (un seul point de contact).
- Si $\Delta > 0$, il y a deux points d'intersection (nous prenons la valeur minimale entre t_1 et t_2 pour avoir le point de contact "frontal" avec notre rayon envoyé).
- Si $\Delta < 0$, il n'y a pas d'intersection.

Voici une représentation pour chacun des cas que nous avons :

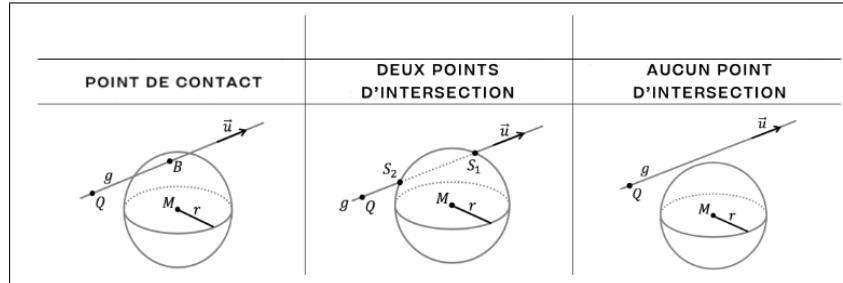


FIGURE 1 – Représentation graphique des différents cas pour Δ

Une fois le calcul des intersections implémenté pour les sphères, voici ce que nous obtenons en essayant différentes configurations (modification de la position, modification de la mise à l'échelle).

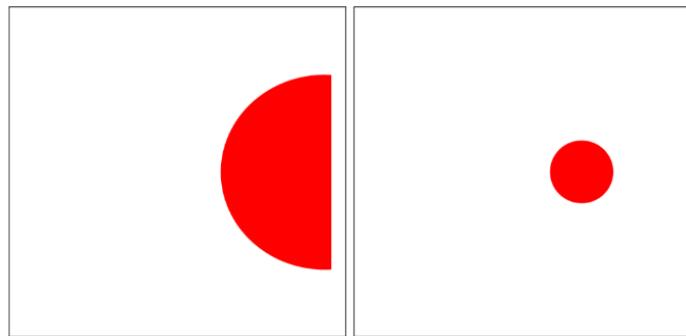


FIGURE 2 – Modification de la position et de la mise à l'échelle

Si on insère deux sphères, on arrive bien à retourner l'objet (ici la sphère verte) le plus proche :

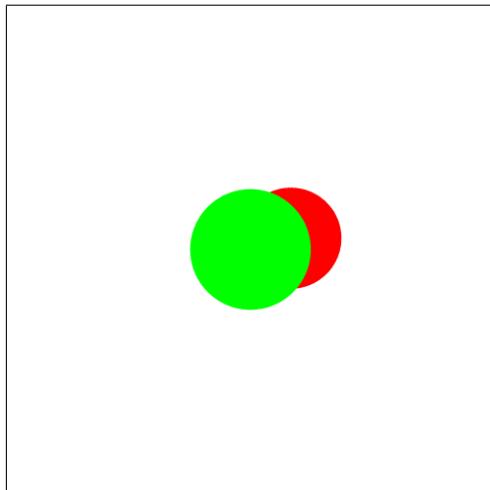


FIGURE 3 – Récupération de l'intersection la plus proche

1.3 Calcul de l'intersection pour un plan

Pour calculer l'intersection entre un rayon et un carré, nous devons tout d'abord récupérer l'intersection avec le plan où se trouve le carré, puis nous pourrons vérifier si l'intersection se trouve bien dans le carré visé.

- Le rayon est défini par : $r(t) = \mathbf{o} + t\mathbf{d}$, où $\mathbf{o} = (o_x, o_y, o_z)$ est l'origine du rayon et $\mathbf{d} = (d_x, d_y, d_z)$ est sa direction.

- Le plan contenant le carré est défini par $(\mathbf{p} - \mathbf{a}) \cdot \mathbf{n} = 0$, où \mathbf{n} est le vecteur normal unitaire au plan ($\|\mathbf{n}\| = 1$) et $D = \mathbf{a} \cdot \mathbf{n}$ est la distance du plan à l'origine (0,0,0).

L'équation du plan peut aussi s'écrire :

$$\mathbf{x} \cdot \mathbf{n} - D = 0$$

Pour trouver l'intersection entre le rayon et le plan, nous remplaçons \mathbf{x} par l'équation du rayon dans l'équation du plan :

$$(\mathbf{o} + t\mathbf{d}) \cdot \mathbf{n} - D = 0$$

En développant, nous obtenons :

$$t = \frac{D - \mathbf{o} \cdot \mathbf{n}}{\mathbf{d} \cdot \mathbf{n}}$$

Nous avons donc 4 cas différents en fonction de t :

- si $t \infty$, cela signifie qu'il n'y a pas d'intersection avec le plan, car le rayon est parallèle au plan
- si t non-défini, le rayon est confondu avec le plan, c'est-à-dire que le rayon est une partie du plan (non parallèle)
- si $t > 0$, l'intersection se trouve derrière la caméra
- si $t < 0$, l'intersection se trouve devant la caméra

Une fois l'intersection avec le plan mis en place, voici ce que nous obtenons :

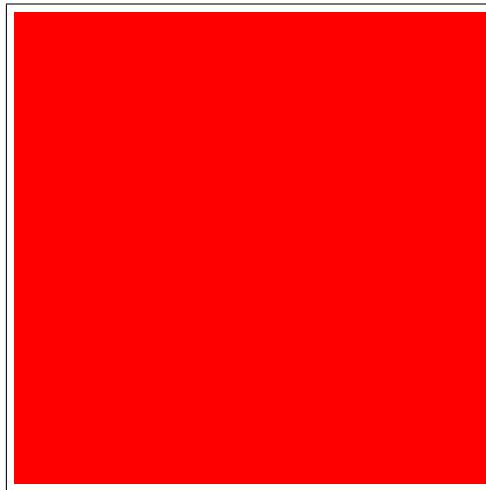


FIGURE 4 – Représentation de l'intersection avec un plan

1.4 Calcul de l'intersection avec un carré

Pour vérifier que le point d'intersection se trouve bien à l'intérieur du carré, nous projetons ce point sur les axes locaux du carré, définis par les vecteurs **up** et **right**, qui représentent respectivement la direction verticale et horizontale du carré. Ces vecteurs permettent de déterminer la position de l'intersection par rapport au coin inférieur gauche du carré, même si celui-ci est orienté de manière arbitraire dans l'espace.

La projection est réalisée à l'aide du produit scalaire, qui mesure la composante du vecteur allant du coin inférieur gauche au point d'intersection, sur chacun des axes locaux. Les résultats obtenus, appelés **u** et **v**, sont ensuite normalisés en divisant par la longueur des vecteurs **right** et **up**.

Enfin, pour valider que l'intersection se trouve à l'intérieur du carré, nous vérifions que **u** et **v** sont compris dans les bornes $[0, 1]$. Si cette condition est satisfaite, alors l'intersection est valide, sinon elle est rejetée. Voici un schéma représentant les projections sur le plan pour détecter la présence du rayon dans le carré :

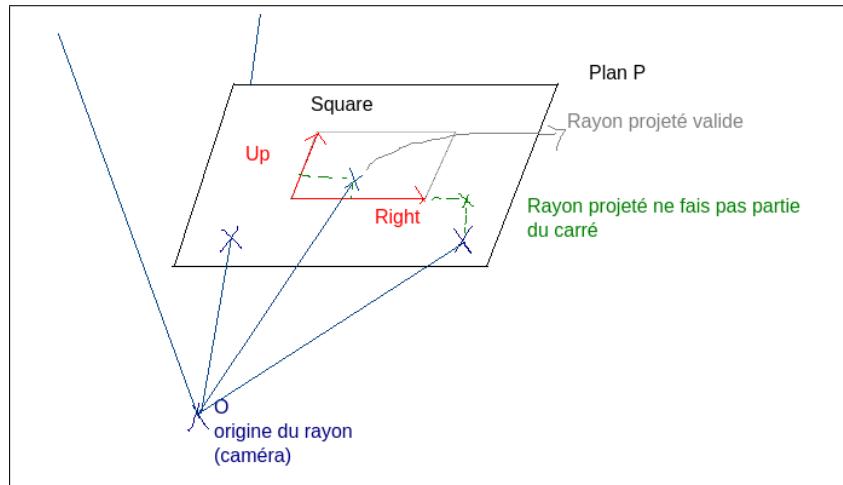


FIGURE 5 – Schéma des projections sur le plan afin de vérifier que le rayon soit dans le carré

Afin de mieux visualiser les intersections obtenues, on construit une scène afin de visualiser le bon fonctionnement de notre fonction et voici ce que nous obtenons :

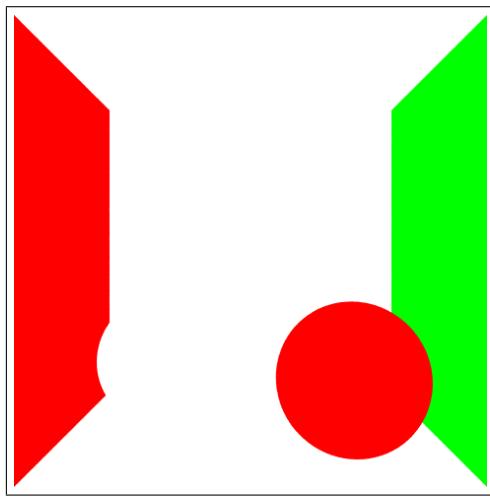


FIGURE 6 – Cration de la scne du raytracing avec les intersections des carrs et sphres

En modifiant maintenant les couleurs des objets de la scne nous avons :

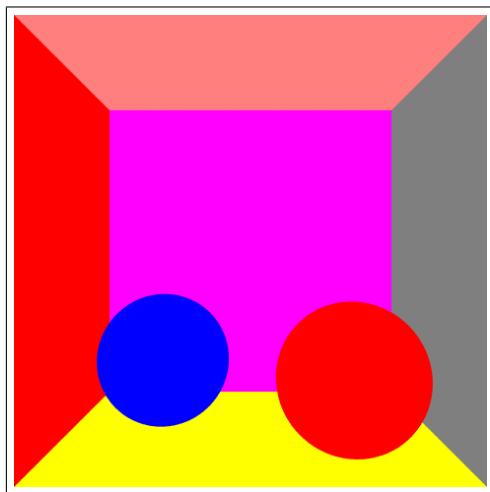


FIGURE 7 – Scnes avec couleurs des objets modifies

2 Phase 2

2.1 Implémentation du modèle de Phong

Le modèle d'éclairage de Phong est un modèle utilisé pour simuler l'interaction de la lumière avec les surfaces en prenant en compte les effets de réflexion diffuse et spéculaire. Il combine trois composantes principales d'éclairage :

- **La composante ambiante** : Elle représente la lumière globale présente dans la scène, qui éclaire uniformément tous les objets. Cette lumière ambiante ajoute un niveau de base de luminosité, simulant l'effet de la lumière réfléchie par l'environnement.
- **La composante diffuse** : Elle décrit la manière dont la lumière est diffusée de façon uniforme sur la surface d'un objet. Cette composante modélise la dispersion de la lumière sur des surfaces mates. La couleur apparente dépend de l'orientation de la surface par rapport à la source de lumière.
- **La composante spéculaire** : Elle simule les reflets brillants et localisés, qui apparaissent lorsque la lumière est réfléchie de manière quasi-miroir. La force du reflet est contrôlée par le paramètre de "brillance", qui ajuste la concentration du reflet.

La composante ambiante se calcule de cette manière :

$$I_a = k_a \cdot I_{sa}$$

Avec :

- I_a l'intensité de la lumière ambiante reçue par la surface
- k_a le coefficient de la réflexion ambiante de la surface (donné par le matériau)
- I_{sa} l'intensité de la lumière ambiante dans la scène

La composante diffuse se calcule comme suit :

$$I_d = k_d \cdot I_s \cdot \max(0, \mathbf{N} \cdot \mathbf{L})$$

Avec :

- I_d l'intensité de la lumière diffuse reçue par la surface,
- k_d le coefficient de réflexion diffuse de la surface (donné par le matériau),
- I_s l'intensité de la source lumineuse,
- \mathbf{N} le vecteur normal à la surface,
- \mathbf{L} le vecteur directionnel vers la source lumineuse.

La composante spéculaire se calcule de la manière suivante :

$$I_s = k_s \cdot I_s \cdot \max(0, \mathbf{R} \cdot \mathbf{V})^\alpha$$

Avec :

- I_s l'intensité de la lumière spéculaire reçue par la surface,
- k_s le coefficient de réflexion spéculaire de la surface (donné par le matériau),
- I_s l'intensité de la source lumineuse,
- \mathbf{R} le vecteur de réflexion de la lumière,
- \mathbf{V} le vecteur directionnel vers l'observateur,
- α le facteur de brillance qui contrôle la concentration du reflet.

En combinant les 3 composantes, voici ce que nous obtenons au fur et à mesure :

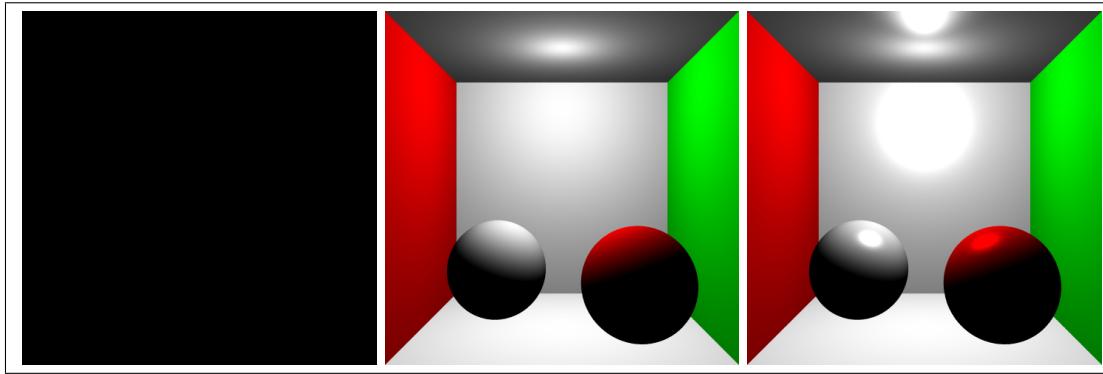


FIGURE 8 – Affichage de la scène en ajoutant les composantes une à une

2.2 Implémentation des ombres dures

Pour implémenter les ombres dures, nous lançons un rayon en direction de la source lumineuse. Si ce rayon rencontre un objet avant d'atteindre la lumière, cela signifie que le point est dans l'ombre, et nous renvoyons alors du noir. Sinon, nous affichons la couleur de l'objet. Voici ce que nous obtenons :

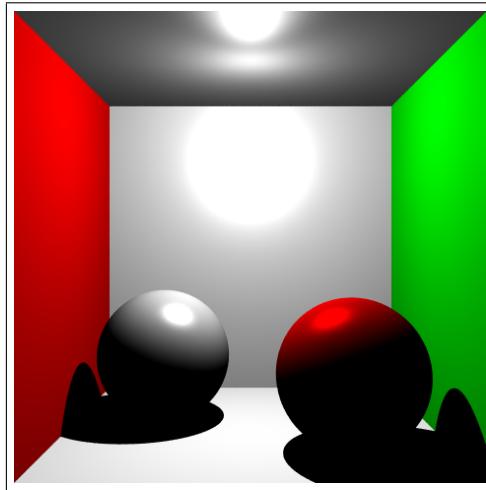


FIGURE 9 – Scène avec implémentation des ombres dures

2.3 Implémentation des ombres douces

Pour implémenter les ombres douces, nous allons simuler une source lumineuse étendue au lieu d'une source ponctuelle. Cela implique de lancer plusieurs rayons en direction de la lumière à partir du point d'intersection sur la surface. Chaque rayon sera testé pour déterminer s'il atteint la source lumineuse ou s'il rencontre un objet intermédiaire. On va récupérer la proportion de signaux reçus par rapport à ceux envoyés et cela sera notre facteur d'ombre utilisé par la suite. Voici ce que nous obtenons :

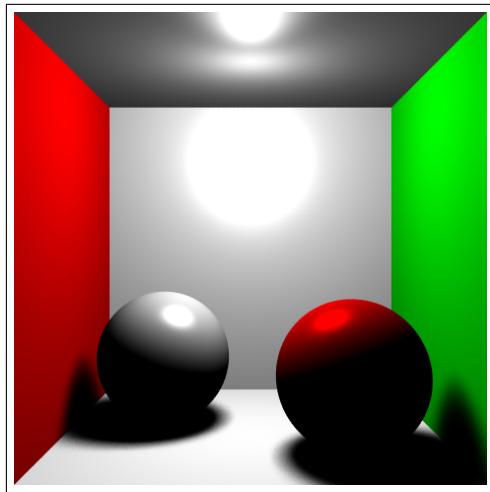


FIGURE 10 – Scène avec implémentation des ombres douces

On voit bien que les ombres sont beaucoup moins "distinctes" et que cela semble plus réaliste.

3 Phase 3

3.1 Application d'un matériau Miroir : Réflexion

Dans cette partie, nous détaillons l'ajout de la réflexion dans le traitement des matériaux miroirs. Cette fonctionnalité permet de simuler des réflexions parfaites sur des surfaces, ce qui est une caractéristique essentielle pour des objets ayant un aspect miroir.

La réflexion est calculée en utilisant le principe suivant : lorsqu'un rayon incident V rencontre une surface réfléchissante au point d'intersection P , le rayon réfléchi R est déterminé en fonction de la normale N à la surface en ce point. La direction du rayon réfléchi est donnée par la formule :

$$R = 2(V \cdot N)N - V$$

avec :

- V , rayon incident
- P , le point d'intersection
- R , le rayon réfléchi
- N , la normal du point d'intersection
- $V \cdot N$, le produit scalaire entre V et N .

Le rayon réfléchi est ensuite lancé depuis une position légèrement décalée $P + \epsilon R$, avec un petit décalage ϵ pour éviter les problèmes d'auto-intersection. Ce rayon est tracé de manière récursive pour calculer la contribution de la réflexion à la couleur finale de l'objet.

Ce procédé permet de simuler avec précision les réflexions dans la scène tout en maintenant un équilibre entre réalisme et performance grâce à une gestion contrôlée du nombre de rebonds de rayons.

Voici ce que nous obtenons une fois que la réflexion est implémentée dans le projet :

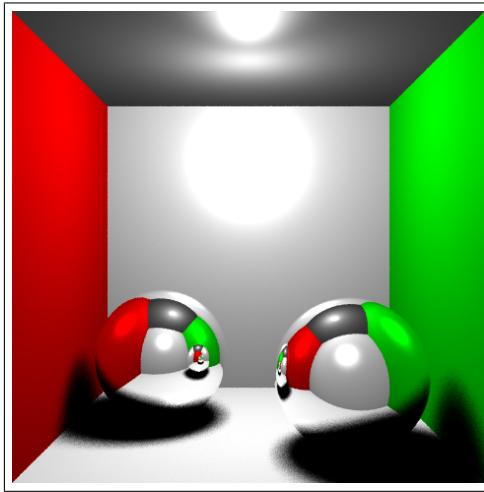


FIGURE 11 – Scène avec deux boules miroirs de réflexion

3.2 Application d'un matériau Vitre : Réfraction

Dans cette partie, nous expliquons l'ajout de la réfraction pour les matériaux transparents comme le verre. La réfraction permet de modéliser la déviation des rayons lumineux lorsqu'ils traversent une interface entre deux milieux d'indices de réfraction différents.

Le calcul de la réfraction repose sur la loi de Snell-Descartes :

$$\eta = \frac{n_i}{n_t}$$

$$\sin(\theta_t) = \eta \sin(\theta_i)$$

Avec :

- n_i , indice réfraction du milieu initial
- n_t , indice réfraction du milieu traversé
- θ_i , angle d'incidence
- θ_t , angle de réfraction

En termes de vecteurs, la direction du rayon réfracté est calculée comme :

$$T = \eta V + (\eta \cos \theta - \sqrt{k})N$$

Avec :

- $k = 1 - \eta^2 (1 - \cos^2 \theta)$
- V , la direction du rayon incident
- N , la normale au point d'intersection
- $\cos \theta$ est le produit scalaire $V \cdot N$

Deux cas doivent être pris en compte :

- Si le rayon entre dans le matériau ($\cos \theta < 0$), la normale N reste inchangée.
- Si le rayon sort du matériau ($\cos \theta > 0$), la normale N est inversée, et les indices de réfraction sont échangés.

Enfin, dans des conditions où $k < 0$ (réfraction impossible) ou selon les probabilités calculées par la formule de Fresnel, une réflexion totale interne peut se produire. Dans ce cas, le rayon réfléchi est traité au lieu du rayon réfracté.

Ce modèle permet de simuler des matériaux comme le verre de manière réaliste en prenant en compte à la fois la réfraction et la réflexion totale interne, offrant un rendu fidèle des objets transparents dans la scène.

Note : Pour éviter une erreur que j'ai eue, j'ai dû appliquer l'approximation de **Schlick** car j'avais un cas où l'on voyait la réfraction d'un objet dans sa propre réfraction. Je l'ai implémentée via cette fonction :

```
static double reflectance(double cosine, double refraction_index) {
    // Use Schlick's approximation for reflectance.
    auto r0 = (1 - refraction_index) / (1 + refraction_index);
    r0 = r0 * r0;
    return r0 + (1 - r0) * std::pow((1 - cosine), 5);
}
```

Cette approximation permet de combiner efficacement la réflexion et la réfraction en gérant les cas limites et en améliorant la précision du rendu.

Voici ce que nous obtenons après avoir mis en place la réfraction :

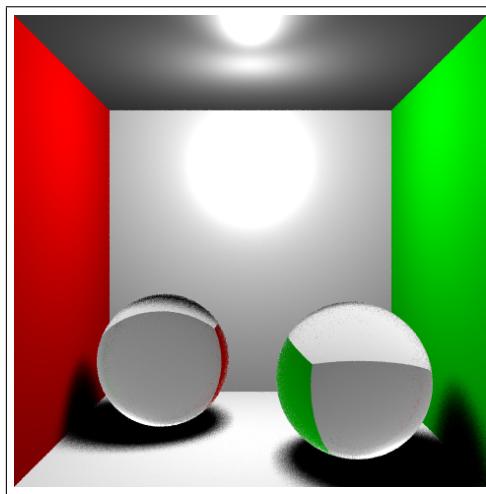


FIGURE 12 – Scène avec deux boules vitres de réfraction

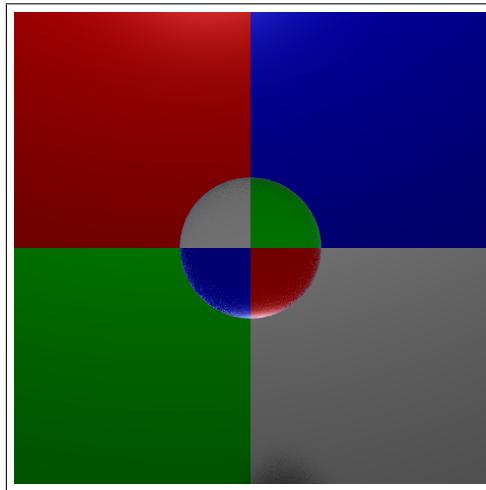


FIGURE 13 – Scène 2 illustrant la réfraction d'une sphère

Voici ce que nous obtenons avec la réfraction et la réflexion dans une même scène :

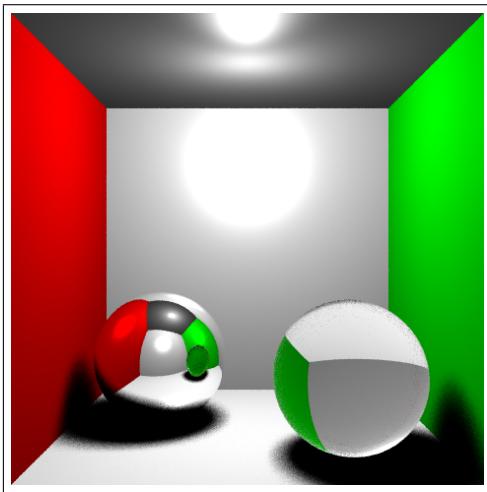


FIGURE 14 – Scène avec de la réfraction et réflexion

3.3 Calcul de l'intersection avec un mesh

Un mesh est constitué d'un ensemble de triangles qui définissent sa surface. Pour savoir si un rayon intersecte un mesh, il est nécessaire de vérifier les intersections avec chacun des triangles qui le composent.

Le processus de calcul de l'intersection avec un triangle repose sur plusieurs étapes :

- **Détection du parallélisme** : La première étape consiste à vérifier que le rayon n'est pas parallèle au plan contenant le triangle. Cela se fait en calculant le produit scalaire entre la direction du rayon et la normale du triangle. Si ce produit est nul, le rayon est parallèle, et aucune intersection ne peut exister.
- **Vérification de la direction du triangle** : Même si un rayon intersecte le plan d'un triangle, il peut arriver que cette intersection se produise sur la "face arrière" du triangle. On vérifie donc que le triangle est orienté face au rayon, en utilisant la normale du triangle et la direction du rayon.
- **Calcul de l'intersection avec le plan** : Si les conditions précédentes sont remplies, on calcule la distance t entre l'origine du rayon et le point d'intersection avec le plan du triangle. Ce point d'intersection est obtenu en résolvant l'équation du plan en fonction de t .
- **Test d'appartenance au triangle** : Une fois le point d'intersection calculé, il reste à vérifier s'il est contenu dans les limites du triangle. Pour cela, on utilise les coordonnées barycentriques, qui permettent d'exprimer un point p en fonction des sommets du triangle (a, b, c) avec

$$p = u \cdot a + v \cdot b + w \cdot c$$

où u, v, w sont des poids barycentriques tels que $u + v + w = 1$. Si l'un de ces poids est négatif, le point p est en dehors du triangle.

3.4 Ajout d'une structure d'accélération : KdTree

Dans cette partie, on s'intéresse à l'ajout d'une structure d'accélération afin d'améliorer la vitesse du rendu lors de la présence de gros maillages. Un KdTree est une structure de données arborescente qui partitionne l'espace en subdivisions hiérarchiques selon les dimensions, permettant ainsi une recherche rapide des intersections en limitant les calculs aux régions pertinentes de l'espace.

Il a été choisi d'implémenter non pas un Kdtree global pour la scène mais un Kdtree pour chaque mesh présent, ainsi, chaque objet dispose de sa propre structure d'accélération, ce qui permet une gestion locale plus efficace des intersections et une meilleure parallélisation des calculs lors du rendu.

3.4.1 Etape 1 du Kdtree : la boîte englobante

La première étape a été de créer la boîte englobante (Englobing Cube) pour chaque mesh. Une boîte englobante est un parallélépipède aligné sur les axes qui encapsule entièrement un objet 3D. Elle est utilisée pour représenter efficacement l'espace occupé par un mesh et simplifier les tests d'intersection.

Pour chaque mesh, les coordonnées minimales et maximales de tous les sommets sont récupérées afin de définir les limites de la boîte (min et max). Cette étape est essentielle pour le partitionnement ultérieur de l'espace dans la construction du KdTree, car elle délimite l'espace à subdiviser.

Donc dans l'étape 1, nous récupérons le point minimal et maximal du mesh associé au kdtree afin de définir l'espace du cube. Et ensuite nous l'affichons afin de bien visualiser le Kdtree. Voici ce que nous obtenons par exemple avec 2 mesh complexes :

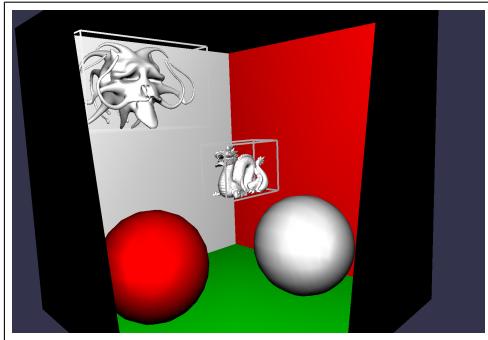


FIGURE 15 – Scène avec les boîtes englobantes des Kdtree

3.4.2 Etape 2 du Kdtree : Création de l'arbre

Une fois chaque boîte englobante créée pour le mesh, c'est l'arbre en lui-même qui est construit. Le processus de construction repose sur une subdivision récursive de l'espace englobé par la boîte, selon un axe choisi (x , y ou z). Cette subdivision a pour but de répartir les triangles du mesh en deux groupes équilibrés, en fonction de leur position médiane le long de l'axe sélectionné.

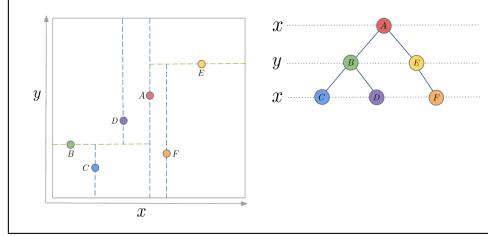


FIGURE 16 – Schéma représentant un kdTree

Le code implémente cette logique en procédant comme suit :

- 1) **Initialisation de la racine** : La racine de l'arbre est définie avec la boîte englobante complète et tous les triangles du mesh
- 2) **Choix de l'axe de subdivision** : Un axe optimal est choisi en fonction de la longueur des dimensions de la boîte englobante, pour garantir une répartition équilibrée
- 3) **Tri des triangles** : Les triangles sont triés selon leurs coordonnées médianes sur l'axe sélectionné
- 4) **Subdivision récursive** : Les triangles sont divisés en deux groupes : ceux à gauche et ceux à droite de la médiane. Deux nœuds enfants sont créés et le processus est répété pour chaque nœud, jusqu'à atteindre une condition d'arrêt (par exemple, un nombre minimal de triangles ou une profondeur maximale, dans notre cas c'est une profondeur maximale).

Une fois le Kdtree implémenté, on a observé une hausse significative des performances pour le traçage des rayons. Par exemple, pour la scène suivante, nous avons 2 meshes complexes avec 871414 et 78966 triangles respectivement (dragon.off et SeaMonster.off) et nous arrivons à réaliser un rendu en seulement 30 secondes !

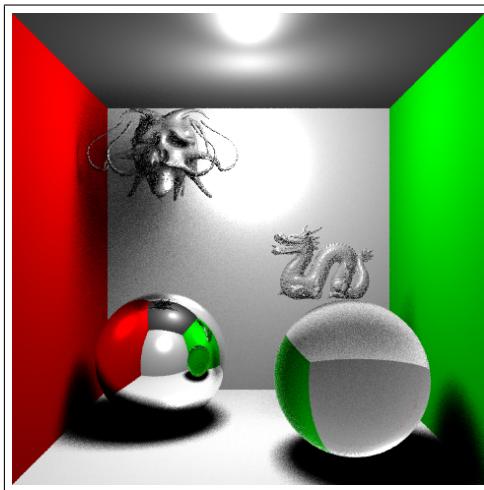


FIGURE 17 – rendu d'une scène contenant 2 mesh complexes avec 871414 et 78966 triangles en 30 secondes

3.5 Bonus 1 : Ajout de textures

Dans cette partie, on s'intéresse au fait d'ajouter des textures aux différents éléments de notre scène : Squares, Circle et Mesh. Le principe d'ajout de texture va différer légèrement en fonction de l'objet suivi, mais va suivre une procédure standard résumée dans ce schéma que nous allons expliquer par la suite :

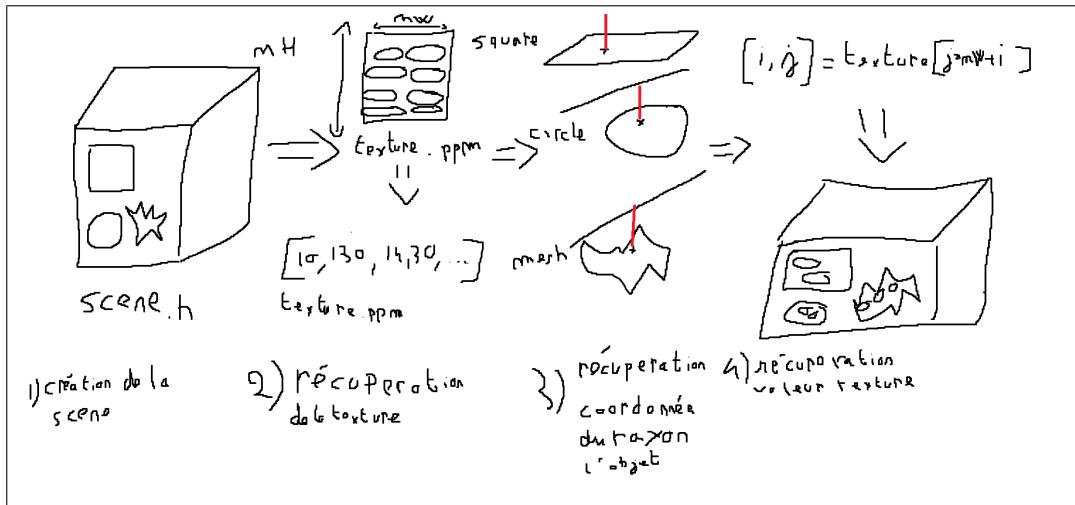


FIGURE 18 – Schéma représentant la procédure d'insertion de texture

Création de la scène

Tout commence lors de la création de la scène, lorsque l'on spécifie les spécifications des mesh (translation, rotation, ect...) on va alors spécifier un matériel contenant le nom de la texture associée au mesh comme ceci :

```
mesh.material = Material("img/sphereTextures/s6.ppm", Vec3(1.0, 1.0, 1.0), Vec3(1.0, 1.0, 1.0),
                           Vec3(0.0, 0.0, 0.0), 16);
```

Récupération de la texture

Une fois le nom du fichier .ppm associé au matériel, toute la partie suivante va être de récupérer l'intégralité des données de la texture. Pour cela, c'est le code des fichiers **image_utils.cpp**/**image_utils.h** qui va entre autres :

- lire le nombre de ligne (nH) et le nombre de colonne (nW) (que l'on définira comme un rectangle de taille nTaille (nH*nW))
- allouer un tableau unidimensionnel (afin de représenter tous les pixels de la texture) de taille 3*nTaille pour faire succéder les 3 composantes (rouge,verte,bleu) de chaque pixel à la suite
- parcourir chaque pixel de la texture, et l'insérer dans le tableau alloué à sa case correspondante ($j * \text{nW} + i$)

Récupération de la coordonnée du rayon

Par la suite, le principe est simple, on récupère la coordonnée du point intersecté par le rayon, ce principe diffère en fonction de la forme intersectée (Cf 3.5.x).

Récupération de la valeur de la texture associé

Une fois toutes les coordonnées du point intersecté récupérées, on va aller chercher le pixel de texture qui correspond à l'intersection, et enfin on l'applique.

3.5.1 Ajout de texture pour un carré

Pour récupérer la coordonnée du point d'intersection sur un carré, l'opération repose sur une méthode simple. Étant donné que le carré partage le même format que notre texture (un rectangle), il suffit d'extraire les coordonnées, u, v de l'intersection. Ces coordonnées normalisées permettent ensuite de localiser directement l'indice du pixel correspondant dans le tableau de la texture (cf 1.4 pour plus de détails).

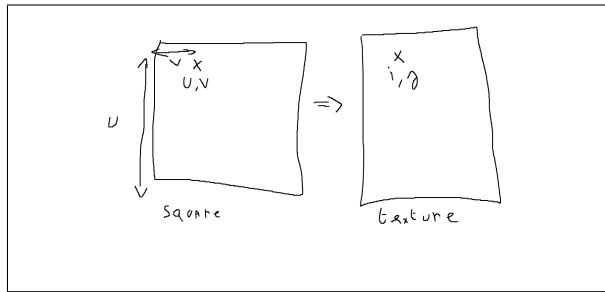


FIGURE 19 – Schéma sur la récupération de coordonées sur un carré

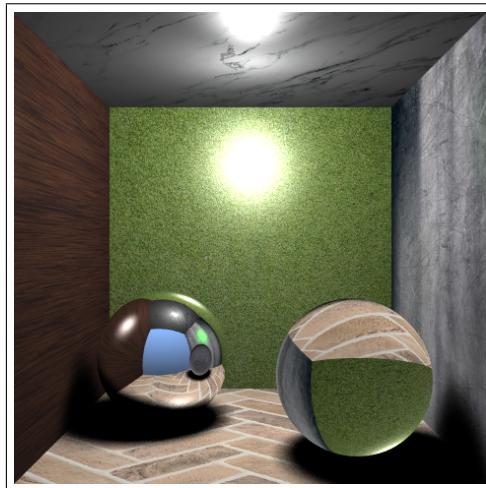


FIGURE 20 – Exemple 1 d'application de texture sur des carrés

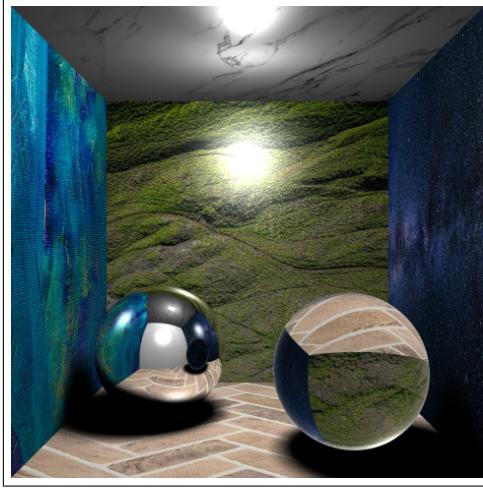


FIGURE 21 – Exemple 2 d’application de texture sur des carrés

3.5.2 Ajout de texture pour une sphère

Pour récupérer la bonne coordonnée du pixel correspondant à une intersection sur la sphère, nous projetons le point d’intersection dans l’espace 2D de la texture à l’aide des coordonnées sphériques. Tout d’abord, le point d’intersection est normalisé par rapport au centre de la sphère. Ensuite, les angles θ et ϕ sont calculés respectivement pour décrire la position en latitude et en longitude sur la sphère. Ces angles sont ensuite convertis en coordonnées (u,v) normalisées, qui sont proportionnelles à la largeur et à la hauteur de la texture. Enfin, ces coordonnées sont mappées à l’image de la texture, et les indices des pixels correspondants sont extraits en tenant compte des dimensions de la texture. Si l’indice calculé dépasse les limites de l’image, une opération de clamp est effectuée pour éviter tout débordement.

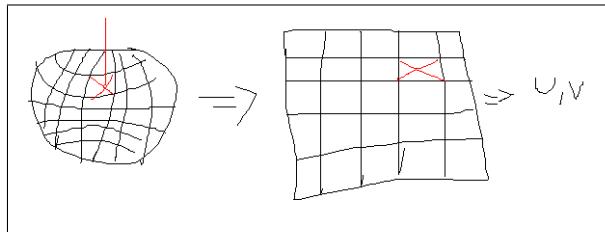


FIGURE 22 – récupération des coordonées u v pour une sphère

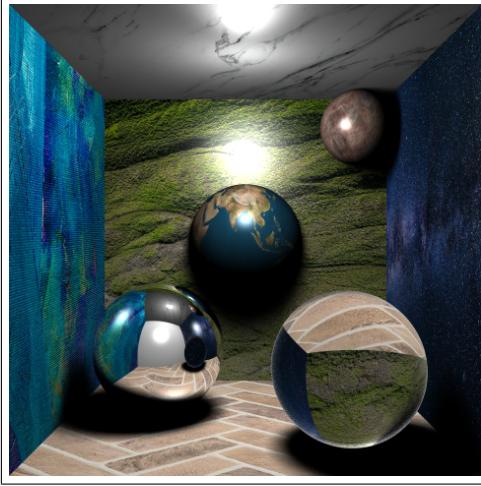


FIGURE 23 – Exemple d’application de texture sur des sphères

3.5.3 Ajout de texture pour un mesh

Dans une section précédente, nous avons expliqué comment calculer l’intersection entre un rayon et un triangle. Cependant, pour appliquer une texture sur le mesh, il est nécessaire de déterminer les coordonnées paramétriques u et v associées à chaque point d’intersection. Ces valeurs sont essentielles pour accéder à la texture et extraire la couleur correspondante au point d’intersection.

Pour calculer u et v , nous utilisons une interpolation barycentrique. Cette méthode repose sur l’utilisation des vecteurs définis par les sommets du triangle et le point d’intersection p :

$$\begin{aligned}\vec{v}_0 &= \vec{b} - \vec{a}, & \vec{v}_1 &= \vec{c} - \vec{a}, & \vec{v}_2 &= \vec{p} - \vec{a}, \\ d_{00} &= \vec{v}_0 \cdot \vec{v}_0, & d_{01} &= \vec{v}_0 \cdot \vec{v}_1, & d_{11} &= \vec{v}_1 \cdot \vec{v}_1, \\ d_{20} &= \vec{v}_2 \cdot \vec{v}_0, & d_{21} &= \vec{v}_2 \cdot \vec{v}_1.\end{aligned}$$

Ces valeurs permettent de calculer le dénominateur commun pour les poids barycentriques :

$$\text{denom} = d_{00} \cdot d_{11} - d_{01} \cdot d_{10}.$$

Les poids barycentriques v , w et u sont ensuite donnés par les formules :

$$v = \frac{d_{11} \cdot d_{20} - d_{01} \cdot d_{21}}{\text{denom}}, \quad w = \frac{d_{00} \cdot d_{21} - d_{01} \cdot d_{20}}{\text{denom}}, \quad u = 1.0 - v - w.$$

Ces coefficients barycentriques u , v et w permettent de déterminer si le point p se situe bien à l’intérieur du triangle (tous les poids doivent être positifs).

Une fois u et v obtenus, ils sont utilisés comme coordonnées paramétriques pour accéder à la texture, récupérer la couleur correspondante, et l’appliquer au point d’intersection.

Voici ce que nous obtenons avec les meshes triangles.off et unit_shpere.off :

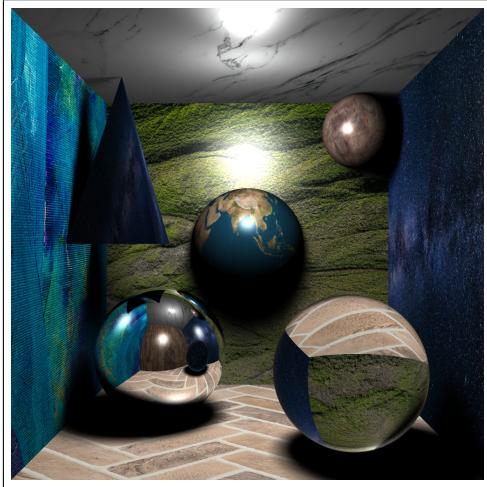


FIGURE 24 – Exemple 1 d’application de texture sur des mesh

Voici ce que l’on obtient avec pyramid.off et tripod.off :



FIGURE 25 – Exemple 2 d’application de texture sur des mesh

3.6 Bonus 2 : Optimisation des calculs matriciels

Dans cette partie, nous aborderons un travail supplémentaire qui a été réalisé afin d'optimiser les temps de calcul.

Lors de la récupération du projet, la première étape était d'inspecter le code pour s'imprégner de ce dernier et c'est à ce moment précis que des erreurs d'optimisation dans le code ont été observées. Dans main.cpp, lorsque nous envoyions des rayons, certaines lignes de code étaient rappelées plus que nécessaire. Ces lignes de code qui s'exécutaient à chaque rayon étaient des calculs lourds de matrice. Cela réduisait énormément les performances de l'application, il a donc été décidé de passer tous ces calculs dans le fichier "MatrixUtilities.h" et de changer la logique d'appel afin de regrouper tous ces calculs lourds de matrice et de les lancer une unique fois. Pour détailler un peu plus tous les calculs qui ont été grandement réduits, on parle d'inversion de matrices qui étaient calculées à chaque rayon. Désormais les calculs de matrices ne sont réalisés uniquement lorsque c'est nécessaire. Après avoir réalisé ce travail, on a gagné considérablement en temps de rendu.

3.7 Bonus 3 : Indicateur de progrès du rendu de la scène

Dans ce bonus, nous allons parler d'une problématique qui a été écartée : la progression du rendu de la scène. Avant lorsque nous lancions un rendu, nous avions uniquement une ligne dans le terminal "Tracing résolution x résolution". Le problème avec cette ligne c'est qu'à cause de cela certains éléments nous étaient inconnus (la progression actuelle du rendu et le temps estimé restant afin d'avoir une idée de la durée totale). Il a été décidé d'ajouter un indicateur de progression. Désormais lorsque l'utilisateur lance un rendu, il va avoir la progression en temps réel du rendu (en %), l'estimation du temps de rendu restant qu'il lui reste et enfin, à la fin du rendu, le temps total du rendu. Tout cela n'affecte en aucun cas les performances du projet et permet d'ajouter plus d'informations à l'utilisateur.

Pour calculer le temps estimé restant, on observe toutes les x secondes la progression du travail venant d'être traité par rapport au travail total restant. Ensuite, on soustrait le temps que l'on vient de calculer au temps total estimé lors du début du rendu.

Voici quelques images illustrant ce nouvel indicateur :

```
RESIZE THE TERMINAL TO NOT HAVE DISPLAY ERRORS
Accessing to the texture : img/planesTextures/vegetation.ppm
[|||||] 99%
Done for img/planesTextures/vegetation.ppm !
Accessing to the texture : img/planesTextures/paint.ppm
[|||||] 99%
Done for img/planesTextures/paint.ppm !
Accessing to the texture : img/planesTextures/galaxy.ppm
[|||||] 99%
Done for img/planesTextures/galaxy.ppm !
Accessing to the texture : img/planesTextures/bricks.ppm
[|||||] 99%
Done for img/planesTextures/bricks.ppm !
Accessing to the texture : img/planesTextures/marbre.ppm
[|||||] 70%
```

FIGURE 26 – Capture d'écran du chargement des textures

```
Press R anytime to run the rendering process !
rRay tracing a 480 x 480 image
[|||||] 19% Remaining Estimated time: 30.9s
```

FIGURE 27 – Capture d'écran de la progression du rendu via la barre de progression

3.8 Bonus 4 : Application d'une normal map sur une surface

Dans cette partie, on souhaite appliquer une normal map à un objet afin d'avoir un rendu plus réaliste de nos surfaces. Pour cela, on va associer le nom d'un fichier .ppm, une normal map au matériel d'un carré (notre surface) afin de l'appliquer. Pour associer la normal map, on va utiliser le même outil développé auparavant **Image Utilities** permettant de récupérer les données d'un fichier .ppm. Lors de l'application du matériel, on va alors allouer tout l'espace nécessaire pour stocker les données de la normal map (la largeur et la longueur de l'image et un tableau unidimensionnel afin de stocker les valeurs de tous les pixels avec leur composante RGB).

Une fois les tableaux alloués, lorsque nous avons un rayon qui touche notre surface, nous appliquons la normale correspondante de la normal map associée à la surface. La normale extraite est ensuite transformée de l'espace tangent à l'espace monde à l'aide des vecteurs tangent, bitangent, et normal (normale de la géométrie). Voici ce que nous obtenons pour la texture et la normal map suivante :

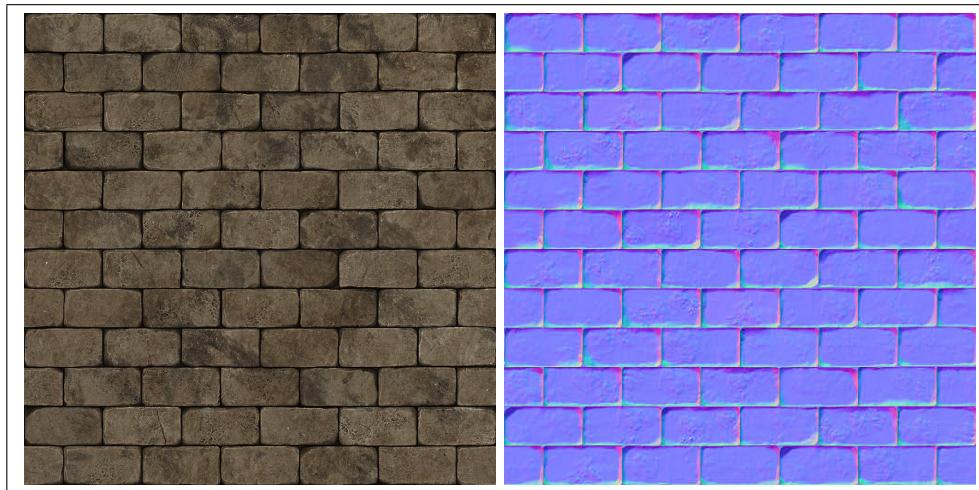


FIGURE 28 – Exemple de texture et sa normal map associée

Voici ce que nous avons avant et après avoir implémenté et activé la normal map dans le matériel de l'objet :

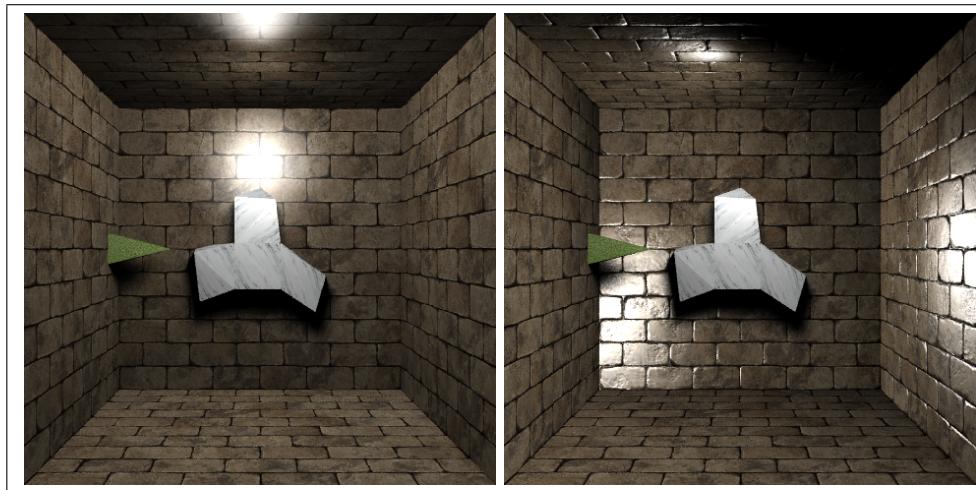


FIGURE 29 – Exemple de texture et sa normal map associée

Références

- [ART1] Peter Shirley, Trevor David Black, Steve Hollasch RayTracing in One Weekend.
<https://raytracing.github.io/books/RayTracingInOneWeekend.html>
- [ART2] Wikiepdia, Intersection (géométrie)
[https://fr.wikipedia.org/wiki/Intersection_\(g%C3%A9om%C3%A9trie\)](https://fr.wikipedia.org/wiki/Intersection_(g%C3%A9om%C3%A9trie))
- [ART3] Stack Overflow, How to display a progress indicator in pure C/C++ (cout/printf) ?
<https://stackoverflow.com/questions/14539867/how-to-display-a-progress-indicator-in-pure-c>
- [ART4] Stack Overflow, barycentric coordinate clamping on 3d triangle
<https://stackoverflow.com/questions/14467296/barycentric-coordinate-clamping-on-3d-triangle>
- [ART5] Stack Overflow, How to get UV coordinates for sphere (Cylindrical Projection)
<https://gamedev.stackexchange.com/questions/114412/how-to-get-uv-coordinates-for-sphere-cylindrical-projection>
- [ART6] Stack Overflow, How can I clear console
<https://stackoverflow.com/questions/6486289/how-can-i-clear-console>